CS 184: Assignment 5—Simple Raytracer

Ravi Ramamoorthi

1 Goals and Motivation

This assignment asks you to write a first simple raytracer. Raytracers can produce some of the most impressive renderings, with high quality shadows and reflections. In fact, you will be using much the same file format as for assignment 2, allowing you to make direct comparisons of images produced using standard OpenGL rasterization and with raytracing.

Raytracers are conceptually very simple. However, the actual implementation effort can be considerable. Therefore, you should start early and proceed through the assignment strictly in the order of the specifications provided. The assignment can be fairly hard to debug. You should strive to make progress incrementally. Start with the simplest functionality (can you render an image with one triangle on the screen?), debug that fully and then proceed. Trying to write the whole thing at once will lead to a mess of undebuggable code.

In this assignment, we recommend but do not require you to work in a group of two (the requirements are only slightly reduced if you work alone, hence the strong recommendation). In general, if you have a partner you worked well with in assignment 4 or otherwise, please work together. However, if you are unable to find a suitable partner and are sure you can handle the workload, it is ok to go it alone.

2 Logistics

Raytracing is well understood and covered in the lecture material, as well as a number of other texts and online materials. At this time, we do not provide any skeleton code for this assignment. Again, your task is to build a real system. For an example of some of the effects you can obtain, you could download public domain raytracers like PBRT or POVray. (You may also want to look at some of the images linked off the assignments webpage.) I have no objection to your looking at the source code in them for inspiration and understanding, but all code you write must be your own. You are of course welcome to reuse code from previous assignments, such as homework 2.

We do provide some test scene files. Download the file *testscenes.zip* which has three test scenes, that are documented and have multiple camera positions you should try. There are also images of these scenes with different camera specifications. Note that these images were created in an OpenGL previewer and are a useful guide, but do not have the sophisticated shading, shadows and reflections, that your raytracer will provide for the same images.

Beyond these initial examples, we do provide an image-based feedback server for this assignment. Logistics, including download of test scenes will be posted on Piazza. As with previous assignments, you are required to provide a link to the feedback server output. Please also note that the feedback server sends rays through the center of a pixel (i.e., at locations 0.5, 1.5, 2.5 and so on rather than at integer values). This is useful for getting an exact match.

Finally, for the purpose of (only) actually writing the output image file, you can reuse any image processing libraries either online or that you have for earlier assignments. The teaching assistants can help in making these resources available online. If you feel this would be too much bother, you can also write out ppm files, and potentially convert them offline (instructions regarding image formats for the feedback server will be posted along with directions). The format for ppm is very simple; just look up portable pixel map on the web. Note that this assignment just requires the ability to write an output file, and does not require OpenGL.

3 Submission

For submission of the final assignment, zip up your source code (plus executable) and place it in the submission directory, as well as *hw5.txt* that provides a link to the feedback server output. Also, include a *README* file that contains any relevant instructions, and a link to a website (on your class or other account) that you should not modify after the due date. (The website is optional but is highly recommended to showcase additional examples and is required for us to evaluate any extra credit). Both partners should submit, indicating in the *README* who their partner is. The *README* can also briefly describe any other particularly interesting aspects of your assignment. It would also help for grading to be honest, noting what works and what doesn't work in the *README*. We may have a demo session where you can explain your program/website. As usual, submission is by running "submit hw5."

For the milestone, you can run "submit hw5milestone." Similar requirements apply, but you would of course have done much less of the assignment (and you need only include a link to the website or a PDF document, not the source code). In particular, I expect that you have completed at least the camera routine, so you can make an image of something, at least a quad. You should also go further, either in beginning to develop the system and parser for the scene geometry, or beginning to code up the ray-surface intersection tests. Your submission should include at least one example image, and a *README* that also briefly describes how/timeframe you intend to complete the remaining assignment. If you're unable to do this or can render only a black screen, please speak to the instructors or GSIs for additional help. (No *hw5.txt* file or feedback server links are required for the milestone). The milestone will count for about 15% of the total grade.

4 Assignment Specification

In general, you should implement a raytracer. The raytracer can be run on the command line with a single argument, that is an input file. All parameters are contained in the input file, whose format is specified below. Your raytracer will parse the input file, reading in geometry, materials, lights, transforms etc. It will then raytrace the scene displaying an image. The file format is rather similar to that in assignment 2, and so you should already have a handle on some logistical issues like how to parse it and so forth. In fact, the assignment is in some sense simpler than homework 2, since you do not need to implement any user interaction.

Finally, ray tracers (especially unaccelerated ones like what you are building) tend to be very slow. You should display some kind of progress indicator to let one see how much of the scene is done (text printed out is fine). Also, for debugging, always start with low resolution images (say 160×120) and make sure things look reasonable before rendering final high resolution (640×480 or higher) versions.

4.1 File Format

The input file consists of a sequence of lines, each of which has a command. For examples and clarifications, see the example input files. The lines have the following form. Note that in practice, you would not implement all these commands at once but implement the smallest subset to debug the first aspect of your raytracer (camera control), then implement more commands to go to the next step and so on. This subsection contains the complete file specification for reference.

- *# comments* This is a line of comments. Ignore any line starting with a #.
- *blank line* The input file can have blank lines that should be ignored.
- *command parameter1 parameter2 ...* The first part of the line is always the command. Based on what the command is, it has certain parameters which should be parsed appropriately.

We now discuss each of the various commands you need to implement, along with the default values to use where appropriate.

4.1.1 General

You should implement the following general commands:

1. *size width height*: The size command must be the first command of the file, which controls the image size.

2. *maxdepth depth*: The maximum depth (number of bounces) for a ray (default should be 5).

3. *output filename*: The output file to which the image should be written. You can either require this to be specified in the input, or you can use a suitable default like stdout or raytrace.tga (you are permitted to use any common image file format for output, that the feedback server can read).

4.1.2 Camera

The camera is specified as follows. In general, there should be only one camera specification in the input file; what happens if there is more than one specification can be left undefined. (You can require the camera command to come before any geometry in the file, although it doesn't seem you really need to require that).

1. *camera lookfromx lookfromy lookfromz lookatx lookaty lookatz upx upy upz fov* specifies the camera in the standard way, as in homework 2. Note that fov stands for the field of view in the y direction. The field of view in the x direction will be determined by the image size. The world aspect ratio (distinct from the width and height that determine image aspect ratio) is always 1; a sphere at the center of a screen will look like a circle, not an ellipse, independent of the image aspect ratio. This is a common convention but different from some previous specifications like the raytracing journal that is linked to in some CS 184 resources.

4.1.3 Geometry

For this assignment, you will worry only about spheres and triangles. These can be specified in a number of different ways, and differ substantially from what you were asked to do for assignment 2. The first set of commands you need to implement are as follows:

1. *sphere x y z radius*: Defines a sphere with a given position and radius.

2. *maxverts number*: Defines a maximum number of vertices for later triangle specifications. It must be set before vertices are defined. (Your program may not need this; it is simply a convenience to allocate arrays accordingly. You can ignore this command [but still parse it] if you don't need it).

3. *maxvertnorms number*: Defines a maximum number of vertices with normals for later specifications. It must be set before vertices with normals are defined. (same discussion as above)

4. *vertex x y z*: Defines a vertex at the given location. The vertex is put into a pile, starting to be numbered at 0.

5. *vertexnormal x y z nx ny nz*: Similar to the above, but define a surface normal with each vertex. The vertex and vertexnormal set of vertices are completely independent (as are maxverts and maxvertnorms).

6. *tri v1 v2 v3*: Create a triangle out of the vertices involved (which have previously been specified with the vertex command). The vertices are assumed to be specified in counter-clockwise order. Your code should internally compute a face normal for this triangle.

7. *trinormal v1 v2 v3*: Same as above but for vertices specified with normals. In this case, each vertex has an associated normal, and when doing shading, you should interpolate the normals for intermediate points on the triangle.

4.1.4 Transformations

You should be able to apply a transformation to each of the elements of geometry (and also light sources). These correspond to right-multiplying the modelview matrix in OpenGL and have exactly the same semantics, just like in assignment 2. It is up to you how exactly to implement them. At the very least, you need to keep track of the current matrix. (Presumably, you can reuse some of the same implementation you did for assignment 2). For triangles, you might simply transform them to the eye coordinates and store them there. For spheres, you could store the transformation with them, doing the trick of pre-transforming the ray, intersecting with a sphere, and then post-transforming the intersection point. The required transformations to implement are:

1. *translate x y z*: A translation 3-vector.
2. *rotate x y z angle*: Rotate by angle (in degrees) about the given axis as in OpenGL.
3. *scale x y z*: Scale by the corresponding amount in each axis (a non-uniform scaling).
4. *pushTransform*: Push the current modeling transform on the stack as in OpenGL. You might want to do pushTransform immediately after setting the camera to preserve the "identity" transformation.
5. *popTransform*: Pop the current transform from the stack as in OpenGL. The sequence of popTransform and pushTransform can be used if desired before every primitive to reset the transformation (assuming the initial camera transformation is on the stack as discussed above).

Note that all of these commands are exactly the same as in assignment 2.

4.1.5 Lights

You should implement the following lighting commands.

1. *directional x y z r g b*: The direction to the light source, and the color, as in OpenGL.
2. *point x y z r g b*: The location of a point source and the color, as in OpenGL.
3. *attenuation const linear quadratic*: Sets the constant, linear and quadratic attenuations (default 1,0,0) as in OpenGL. By default there is no attenuation (the constant term is 1, linear and quadratic are 0; that's what we mean by 1,0,0).
4. *ambient r g b*: The global ambient color to be added for each object (default is .2,.2,.2).

This is slightly different from the specification in assignment 2. Note also that if no ambient color or attenuation is specified, you should use the defaults (you may have used black as a default in assignment 2; here the defaults are specified). Finally, note that here and in the materials below, we do not include the *alpha* term in the color specification.

4.1.6 Materials

Finally, you need to implement the following material properties.

1. *diffuse r g b*: specifies the diffuse color of the surface.
2. *specular r g b*: specifies the specular color of the surface.
3. *shininess s*: specifies the shininess of the surface.
4. *emission r g b*: gives the emissive color of the surface.

4.2 Camera

The first step is to implement the camera model. This step should be done when you submit your milestone. The user should be able to specify the camera, and you should test using a simple scene. You should in particular, include the images corresponding to the first test scene which shows a single quad (for now, it's acceptable if you code up a simple ray-quad intersection test and don't worry about shading). For this part of the assignment, you will need to know how to set a camera, and how to generate corresponding rays for each pixel. Get this part completely debugged before proceeding further.

4.3 Ray-Surface Intersection tests

Now, you should implement the core of your raytracer, which are the ray-surface intersection tests, in this case for triangles and spheres. You should debug separately with each primitive, making sure things work as expected. You could try images of the second test scene of a dice (from each of the camera positions specified)[1]. Next, you should implement transformations, allowing the user to specify transformed geometry. You might want to try the third test scene (the table with ellipses and spheres). Again, shading is not yet important, but you should be sure the core ray-surface intersection tests for geometry are debugged.

4.4 Lighting and Shadows

Next, you should implement shading. For this, simply implement the similar shading model as in homework 2. In particular, the color at each point is given by

$$I = A + E + \sum_i V_i \frac{L_i}{c_0 + c_1 r + c_2 r^2} \left(D \max(N \cdot L, 0) + S \max(N \cdot H, 0)^s \right), \tag{1}$$

where I is the final intensity, A is the ambient term, E is the self-emission, and the diffuse D and specular S are summed over all lights i with intensity L_i. N is the surface normal, L is the direction to the light, H is the half-angle, and s is the shininess. For ray tracers, there is an additional term V_i which is the (for now binary) visibility to the light, corresponding to shadows. You should cast a shadow ray to all lights at the intersection point to determine if they are visible (determine V_i). If visible, we simply compute the diffuse contribution as well as the specular contribution. We also include an attenuation model, new from homework 2, corresponding to traditional OpenGL (the attenuation model applies only to point lights, not directional lights). c_0, c_1, c_2 are the constant, linear and quadratic attenuation terms, while r is the distance to the light. Physical point lights have $(c_0, c_1, c_2) = (0, 0, 1)$ while the default (no attenuation) is $(c_0, c_1, c_2) = (1, 0, 0)$.

4.5 Recursive Ray Tracing

Next, you should implement a recursive ray tracer for mirror reflections. Extra credit will be given for also including refractions (you will need to augment the file format accordingly). The simplest way of doing it is to shoot a single ray in the mirror direction, weighting its contribution by the specularity or S. Since this reflected ray may spawn additional reflections, the tracing is recursive, with the maximum depth of the ray tree controlled by the maxdepth parameter.

4.6 Acceleration Structure (not required for students working alone)

Ray tracing has historically been a slow process, and the scenes you are provided therefore have pretty simple object geometry. To get raytracing to work on more complicated scenes, some type of ray tracing acceleration structure is required. Therefore, this assignment requires that you implement some sort of acceleration scheme. Note that this refers to a conventional geometric acceleration structure (such as axis-aligned bounding boxes or kd-trees or uniform/non-uniform grids), not optimizations such as parallelization or using hardware.

[1] Shading is not too important for now. I've just used a separate ambient color for each face. It's not clear it makes sense for your raytracer to follow this trick; you might imagine adding a command to just explicitly set the color for each face without worrying about lighting for now.

(This can be difficult; at least 90% of the credit will be given for the core components above, but if you want a perfect score, you must do this part; however, please do it last and you will lose only a very few points if you do not do it. This part is also only required for groups of two; not for students working alone.).

We do not specify what acceleration scheme you should use, and any method is fine. Perhaps the simplest is to just implement a regular uniform grid; each grid cell is a small cube and holds any geometry that intersects that grid cell (so some geometry could be duplicated). When tracing a ray, you go to each grid cell in order, and intersect only with the geometry in that cell. If a ray doesn't intersect some grid cells, they can be avoided altogether. The trick is to find out for each sphere/triangle, which grid cells it intersects, and store them appropriately, and then to augment the ray marcher to know when it hits the next grid cell. You will need to play with grid resolution, perhaps starting with something like $5 \times 5 \times 5$.

If you do get this working, please document the algorithm and speedup in your writeup *README*. You may need new test scenes with hundreds of objects to really document and test the speedup (the feedback scenes also include one or two examples for this purpose). Perhaps it is simplest to find *.off* files online which have a very simple geometry format. You may already have used some of them for homework 3. You could either convert them to your raytracer format, or get the the raytracer to read that format directly.

Again, note that this part will not be judged too harshly, and will lead to only a small deduction if not implemented. However, it is important to do a complete raytracer, and I hope that at least some groups will go forward. We will not be picky; we just require some effort at acceleration.

4.7 Extra Credit

Up to 10-15 points of extra credit will be given for additional features and/or the best images produced by your ray tracer. Feel free to exercise your creativity here and go overboard. Obvious ideas are more primitives, better acceleration structures, soft shadows from area lights, Monte Carlo sampling for handling complex lighting and materials, and so on. You can look at some of the best assignments from previous years, as also the final project extensions for ideas.